下载APP

课堂答疑（一）｜前置篇、C 核心语法实现篇问题集锦

2021-12-31 于航

《深入C语言和程序运行原理》 课程介绍 >

讲述：于航

时长 08:53 大小 8.15M

你好，我是于航。

看到这里的你，应该已经完成了本课程前两个模块的学习。随着课程的不断推进，我陆续收到了很多反馈。很高兴看到你在评论区积极留言，和大家一起讨论思考题。并且，还有很多同学提出了一些非常有意义的问题。那么，在继续学习后面更深入的内容之前，让我们先暂缓脚步，从问题出发，进行一次整体性的回顾。

领资料

在这一讲中，我会以加餐的形式，为你剖析一些值得讨论的思考题，以及你们提出的有代表性的问题。

问题一：在 01 讲 的最后，我留给你的问题是：在这一讲第一部分的 C 代码实例中，我们为何要给函数 findAddr 添加 `static` 关键字？不添加这个关键字的话，程序是否可以

编译运行？

这里，我将那段代码简化了一下，只提取出与问题相关的部分，放到了下面。因此，问题也变成了：对于下面这段代码来说，将函数 foo 定义中使用的 `static` 关键字去掉，是否会影响程序的正常编译和运行呢？

复制代码

```
#include <stdio.h>
inline static int foo(void) {
  return 0;
}
int main(void) {
  printf("%d", foo());
  return 0;
}
```

实际上，如果你能够在 godbolt 上快速实践一下，就会发现：在默认情况下（没有使用任何优化参数），编译器会报出类似 “error: ld returned 1 exit status” 的链接器错误；而在使用 “-O1” 及以上优化参数的情况下，编译器则可以正常编译。那么，为什么会这样呢？

实际上，虽然我们在课程中会使用 C17 标准对代码进行编译，但上述代码中使用的 `inline` 关键字却来源于 C99 标准。对于编译器来说，在这个标准下，标注了 `inline` 关键字的函数**意味着函数的定义仅用于内联**。而当编译器在低优化等级下不选择将函数进行内联处理时，便会尝试去寻找一个可以进行链接的函数实现。

关于这一点，我们也可以从 C 标准中得到印证。在这个链接中， C17 标准文档的 91 页，第 6.7.4.7 小节对 `inline` 关键字进行了总结，你可以参考。

到这里，解决这个问题的方法就变得十分清晰了。通常，我们可以使用以下几个办法：

1. 去掉 `inline` 关键字。由于该关键字通常只作为编译器的 hint，因此在使用优化的情况下，基本不会影响编译结果；
2. 使用 `static inline`，为函数提供 internal linkage。编译器总是能够使用在当前编译单元内具有静态链接的函数；

3. 使用 `extern` 为函数提供对应的 external definition。但这种方式与定义一个普通函数没有区别，从标准上来看，不符合内联的规则，即 inline definition。

另外，你也可以考虑使用 GNU C89 标准进行编译（-std=gnu89），以采用旧式的内联处理机制。但这种方式会让我们无法使用 C 语言的新特性，所以我并不推荐。

问题二：在问题一的例子中，为什么使用 `inline static` 而不是 `static inline` 呢？

这是一个非常棒的问题。实际上，对于编译器而言，这两种写法都可以正常工作。但不同点在于，对于 C17 以前的标准来说，“声明标识符（Declaration Specifier）”是可以按照任意顺序摆放的。而在 C17 及以后的标准中则规定，声明标识符中的 “Storage-class Specifier” 应该被放置在各定义的最开头处，这其中便包含有 `static` 关键字。而对于其他形式，则会被视为过时的写法。但实际上，考虑到向前兼容，现代编译器都还支持这种旧式写法。

问题三：有同学问到，为什么我们在文章代码中使用的内联汇编，与通过 C 代码编译而来的汇编，两者在风格上有很大差异？比如，对于同一个机器指令来说，两种风格在源操作数与目的操作数的使用顺序上竟然是完全相反的。

没错，这是一个相当细心的同学。实际上，对于 x86 汇编语言来说，目前工业界主要有两种不同的代码编写风格，即 Intel 风格和 AT&T 风格。其中，前者被广泛使用在 Windows 操作系统及相关工具链上，而后者则被广泛使用在 “Unix 世界”中（因为 Unix 是从 AT&T 贝尔实验室创建的）。因此，不同的代码风格便也对应着不同的代码编写方式。

这里，我把它们之间的一些主要区别，通过具体的例子进行了描述，并整理在了下面的表格中供你参考：

	AT&T	Intel
参数顺序	movl $5, %eax	mov eax, 5
参数大小	addl $4, %esp	add esp, 4
特殊标记	立即数使用 “$”，寄存器使用“%”	由汇编器自动推导
有效地址	movl 0xff(%ebx, %ecx, 4), %eax	mov eax, [ebx + ecx * 4 + 0xff]

极客时间

问题四：有同学在实践时发现，我在 02 讲 中介绍的一段本无法被正常编译的代码，却可以在 Clang 编译器下被正常编译。这里，为了方便你观察，我将这段代码直接放到了下面：

复制代码

```
1  #include <stdio.h>
2  int main(void) {
3    const int vx = 10;
4    const int vy = 10;
5    int arr[vx] = {1, 2, 3};  // [错误1] 使用非常量表达式定义定长数组；
6    switch(vy) {
7      case vx: {  // [错误2] 非常量表达式应用于 case 语句；
8        printf("Value matched!");
9        break;
10     }
11   }
12 }
```

正如代码中第 5 行和第 7 行注释所描述的那样，我们无法用非常量表达式（值）来作为定长数组的长度参数，或是直接用于 case 语句中。而这段代码之所以能在 Clang 中被正常编译，则是由于编译器通常会竭尽所能地去编译用户提供的代码。

因此，在这个例子中，Clang 会在默认情况下使用名为 “gnu-folding-constant” 的 GNU 扩展，来处理代码中不符合 C 标准的用法。该扩展可以在代码需要的地方，将非常量表达式转换为常量表达式使用。但实际上，这并非 C 标准的内容。

为此，我们可以通过在编译代码时指定 “-pedantic-errors” 选项，阻止编译器对扩展能力的使用。该选项会在编译器使用扩展时发出相应的错误警告。而 GCC 与 Clang 这两个编译器均支持该选项。

问题五：学习 C 语言，真的有必要了解汇编吗？

相信这是一个困扰很多 C 语言初学者的问题，这里我来谈一谈自己的理解。

首先，这个问题是没有标准答案的。C 语言被创造出来的目的之一，就是抹平汇编语言在编码上的差异。通过这种方式，人们可以做到代码的一次编写，多次编译。所以，对于学习或编写 C 代码来说，实际上是不需要了解汇编语言的。编译器会以最好的方式，帮助你将 C 代码转换成对应的机器代码。

但如果是这些情况：在你的应用场景中，单纯使用 C 代码无法完成相关任务（比如需要使用“内联汇编”的场景），或者你对 C 程序的性能优化有着极致的追求，又或是你想对程序的运行细节有更多的理解……那么，适当学习汇编语言与计算机体系结构的相关知识便是必要的。

总而言之，是否需要学习汇编语言，还是要看学习者自身的目的。编程语言是一种工具，学习它首先是要做到能用起来，在此基础上，再根据每个人的不同需求，向不同方向深入。

问题六：`leave` 指令在“清理栈”时会将相关内存清空（置零）吗？

这个问题非常好。一般来说，我们在编码过程中提及所谓“清空”或“清理”时，大部分人第一时间都会认为是将某个量重置为对应的初始值，这其中可能包含有将某个值置为 0 的过程。但实际上，对于 `leave` 指令来说，它仅会在执行时修改寄存器 rsp 和 rbp 的值，但并不会对“残留”在栈中的数据（比如前一个栈帧）做任何处理。你可以通过下面这段代码来验证这个结论：

复制代码

```
#include <stdio.h>
void foo(void) {
  int x;
  printf("%d\n", x);
  x = 10;
```

```
6 }
7 int main(void) {
8   foo();
9   foo();
10  return 0;
11 }
```

因此，对于一个程序来说，相关寄存器中的值便能够完全表示这一时刻该程序的实际运行状态。在接下来的 12 讲中，你将会看到，setjmp 与 longjmp 函数是如何通过保存寄存器的值来恢复函数的执行状态的。

问题七：有同学问到，对于某些简单的算数运算逻辑，编译器为什么会使用 lea 指令，而非 mov 指令来实现呢？

实际上，对于某些特定格式的简单计算，使用 lea 指令可以获得比 mov 指令更好的性能。比如，来看下面这段 C 代码：

复制代码

```
1 int foo(int n) {
2   return 4 * n + 10;
3 }
```

对于这里的 foo 函数来说，编译器可以选择使用 mov 指令按照下面的方式来实现：

复制代码

```
1 mov eax, edi
2 sal eax, 2
3 add eax, 10
```

当然，也可以使用 lea 指令，以一种更加精简的方式来实现：

复制代码

```
1  lea eax, [10+rdi*4]
```

可以看到，使用 lea 指令实现这段代码，可以让程序少执行两条机器指令，从而进一步提升程序的运行时性能。

在 x86 体系中，指令参数可以使用的一种内存地址形式为 [base + scale x offset + displacement]。其中，displacement 必须为 32 位有符号格式的立即数，scale 的可选值为 1、2、4 或 8，base 和 offset 可以由寄存器或者立即数组成。因此，通过这种方式，lea 指令就能够被用于实现多种基本的算术运算，并优化程序性能。

问题八：C 语言可以实现类似 C++ 的函数重载吗？

这个问题很好，但遗憾的是，C 语言并没有提供用于实现多态的相关特性。不过在 C11 之后，借助泛型宏 _Generic，我们也可以在一定程度上实现类似的功能。具体你可以参考下面这段示例代码。而关于这个宏的详细使用方式，可以参考这个链接。

复制代码

```
#include <stdio.h>
#define foo(N) _Generic((N), \
  double: food, \
  default: fooi, \
  float: foof)(N)
int fooi(int n) { return n; }
double food(double n) { return n; }
float foof(float n) { return n; }
int main(void) {
  printf("%d", foo(1));
  return 0;
}
```

好了，今天的答疑就到这里，感谢提出问题的各位同学。如果你还有其他问题，欢迎在评论区与我讨论，或者进入课程专属的微信群（进群入口），与其他小伙伴一起沟通交流。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 3 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 10 | 标准库：字符、字符串处理与数学计算

下一篇 大咖助场 | LMOS 说 C 语言（上）：为什么说 C 语言是一把瑞士军刀？

更多课程推荐

精选留言 (3)

写留言

ZR2021
2022-01-01

老师强大！！！

1

sugar
2022-01-03

请教于航老师一个问题，我的日常开发程序语言主要是JavaScript和C/C++，这个技术栈与老师应该比较类似。在js中，我们在运行时console.log可以打出任何一个代码中可见的变量的具体值，无论它是一个number、string这样的简单类型，还是各种类嵌套定义出的超

级复杂的数据结构，这个能力在阅读其他人写的代码时很有帮助：我不确定一段程序中的某个符号代表的对象究竟是什么结构的时候，我可以直接去运行时把它log出来，尤其是...

展开

1

Fan

2021-12-31

Good